C++ LIMIT ORDER BOOK

Andreas Mitillos
MSc Mathematical Finance
University of Warwick
andreas.mitillos@warwick.ac.uk

Navpreet Minhas
MSc Mathematical Finance
University of Warwick
navpreet.minhas@warwick.ac.uk

Danfei Shao
MSc Mathematical Finance
University of Warwick
danfei.shao@warwick.ac.uk

Britney Toroitich
MSc Mathematical Finance
University of Warwick
britney.toroitich@warwick.ac.uk

Yuheng Wang
MSc Mathematical Finance
University of Warwick
yuheng.wang@warwick.ac.uk

March 12, 2025

1 Introduction

With the rise of financial exchanges and increasing trading activity, efficient trade execution has become essential. The Limit Order Book (LOB) is a system that organises buy (bids) and sell (asks) orders, sorting bids in descending order and asks in ascending order, and matching them based on Price-Time priority. As digital trading advances and transaction volumes grow, high-speed order execution is increasingly important.

This project explores the structure and mechanics of a Limit Order Book using C++ and an Object-Oriented Programming (OOP) approach. While an efficient matching algorithm is a key focus, the primary objective is to develop a deeper understanding of order book mechanics, matching processes, and fundamental OOP principles.

A limit order book is a real-time data structure that facilitates trade execution by maintaining bid and ask orders. Traders specify a maximum price for bids and a minimum price for asks, with trades occurring when a buy price meets or exceeds a sell price.

The project simulates a fully functional LOB with support for market orders, limit orders, stop-market orders, and stop-limit orders. Interaction with the system is achieved via a Command Line Interface (CLI), providing a structured environment for order management and execution.

2 Implementation

A limit order book manages bid and ask prices, representing buy and sell orders at various price levels. The execution of trades within a limit order book follows two fundamental principles: Price-Time Priority, and First-In-First-Out (FIFO) Execution.

The Price-Time Priority rule dictates that orders with the most competitive prices are executed first, i.e., the highest bid price is matched with the lowest ask price. When multiple orders exist at the same price level, FIFO Execution ensures that the earliest placed order is executed first. This prevents newer orders from taking precedence over older ones at the same price level.

Once an order is placed, the order book is automatically updated, prioritising execution based on price before considering the order's placement time. When a trade occurs, the oldest order at a given price level is fully or partially filled before newer orders at the same price.

For stop orders, execution follows the same Price-Time Priority and FIFO principles but only after being triggered. A stop order remains inactive until the market reaches a specified price level (the stop price). Once triggered, it is treated

as either a market or limit order, depending on the order type specified. The user defines this threshold, and once the market price crosses it, the stop order is converted accordingly.

2.1 Object-Oriented Approach

2.1.1 The Order Class

The `Order` class represents an individual order in the limit order book, encapsulating its attributes and behaviours. It supports multiple order types, including market orders, limit orders, stop-market orders and stop-limit orders. The structure of the class is illustrated in Figure 2.

Figure 1: Order Class Overview

Each order type is instantiated using a dedicated constructor. The class primarily serves as a data structure for storing key order details, including whether a stop order has been triggered and whether it was originally a stop order, the order type, the limit price for limit and stop-limit orders, the stop price for stop-orders, the filled quantity representing the number of shares already executed and the side of the order book (buy or sell).

2.1.2 The Order Book Class

The `OrderBook` class serves as the core component of the program, responsible for handling the execution of orders while adhering to the principles of an order book, specifically FIFO (First-In-First-Out) and Price-Time Prioritisation. This class also maintains the storage of active orders, ensuring efficient management and retrieval. The structure of the class is demonstrated in Figure 2.

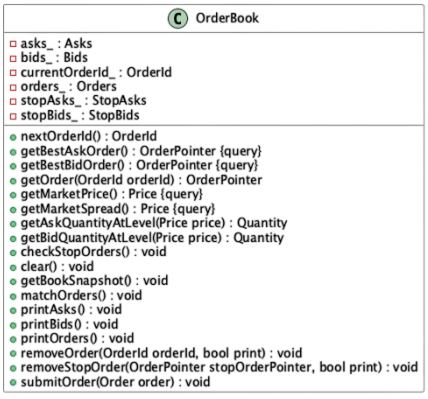

Figure 2: Order Class Overview

Order Book Class Design and Implementation The class relies on several key data structures to efficiently maintain and process orders. The primary data members include: ask and bid maps storing active limit orders and stop ask and stop bids maps storing active stop orders. All these maps are implemented using ordered maps (`std::map`) where the key represents the limit price level for limit orders, and stop price level for stop orders. Each price level in the map stores a list of order pointers (`std::list<std::shared_ptr<Order»`). This list maintains the time-based ordering of orders, with newer orders being appended to the end. This ensures FIFO compliance within each price level. To facilitate efficient order retrieval and execution, bids (buy orders) are stored in descending order (highest to lowest price) and asks (sell orders) are stored in ascending order (lowest to highest price). This structure allows efficient insertion of new orders in $\mathcal{O}(\log n)$ time, leveraging properties of `std::map`. The use of `std::list` within each price level enables $\mathcal{O}(1)$ insertion at the end and FIFO execution.

Memory Management and Pointer Handling A critical design decision consisted of how orders are stored and referenced. Since the program is executed through a Command-Line Interface (CLI), orders are typically created by handler methods within the CLI Handler class. This presents a memory management challenge: if raw pointers were used, they would reference objects that go out of scope once order creation completes, leading to dangling pointers and undefined behaviour. Though heap allocation could be used, it was avoided to avoid potential scalability issues and memory leaks. Instead, the implementation utilises shared pointers (`std::shared_ptr<Order>`). This ensures that an order is not deallocated while still referenced by the `OrderBook`, preventing memory access violations.

Order Matching and Stop Order Execution and Triggering The `OrderBook` class is responsible for order matching and stop order triggering, which will be discussed in detail later in the report. The CLI Handler maintains an instance of the OrderBook class, which purely handles submission and execution logic.

2.1.3 The Command Line Interface (CLI) Handler Class

In a real-world trading system, this program would function as an API that provides access to an order book's functionality, allowing traders to interact with it programmatically. Typically, traders would access order book data through a Graphical User Interface (GUI) rather than a Command-Line Interface (CLI). However, since GUI development is outside the scope of this project, a CLI was implemented to enable user interaction with the limit order book.

This functionality is encapsulated in the `CLIHandler` class, which serves as the interface between the user and the `OrderBook`. The interaction process follows these steps:

1. An instance of the `OrderBook` class is created,
2. An instance of `CLIHandler` is initialised, receiving a reference to the order book, and
3. The `CLIHandler` listens for user input, processes the entered command, and directs it to the appropriate handler

The structure of the `CLIHandler` class is illustrated in Figure 3.

Figure 3: CLIHandler Class Overview

The `CLIHandler` class is responsible for listening for user input, parsing and tokenizing commands, validating user input and executing the appropriate command handlers. At the core of this implementation is the `run` method, which continuously listens for user input. Upon receiving a command, the following steps are executed:

1. The command string is passed to the `parseArguments` method, where it is tokenized by splitting the string into a vector of substrings, using white-space as the delimiter. Leading and trailing spaces are trimmed,
2. The first element of the vector is examined to determine if it is a recognised command,
3. The number of arguments is checked against the expected number of parameters for this given command,
4. If the command and arguments are valid, the respective handler method is called,
5. The handler method performs additional argument validation (e.g., checking data types and sensible price and quantity values), and
6. Depending on the command, the system either creates a new order or display relevant order book information to the user

To ensure the CLI functions correctly, several validation methods are implemented. These methods ensure that inputs conform to the expected data types, commands include the correct number of arguments and orders are created only when valid parameters are provided.

2.2 Market Orders

Market orders execute immediately at the best available price, prioritising speed over price. Unlike limit orders, they are not stored in the order book but matched against existing limit orders until the required quantity is filled. Execution efficiency depends on market liquidity at the time of submission. If sufficient liquidity is available, the order is fully filled; otherwise, it is partially executed. Following execution, the system runs the stop order checking and matching algorithms. A visual representation of the market order submission process is shown in Figure 4.

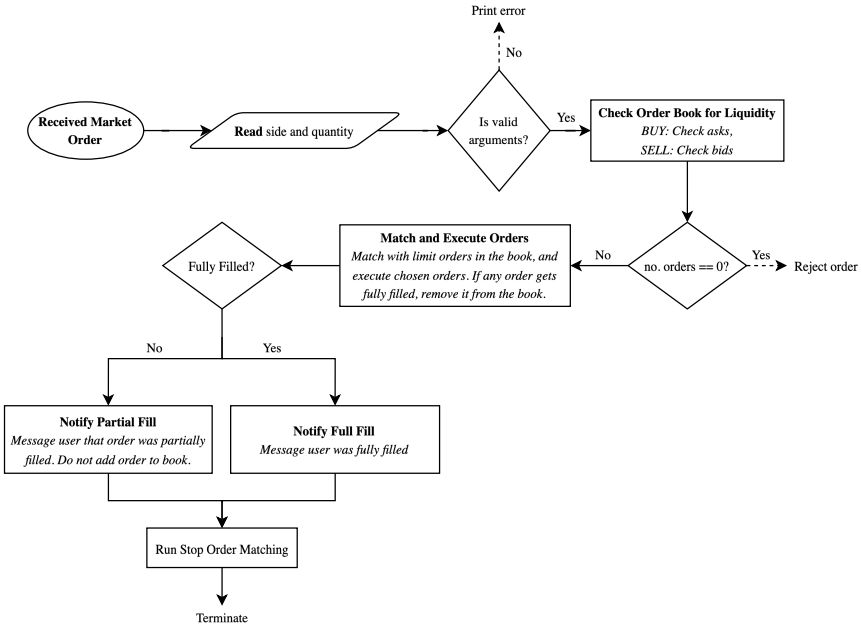

Figure 4: Market order submission process.

2.3 Limit Orders

Limit orders allow traders to specify a maximum (for buys) or minimum (for sells) execution price, ensuring price control but not immediate execution. They remain in the order book until matched with a market order, another limit order, or manually cancelled. Upon submission, the matching algorithm checks for price crossings, followed by the stop

order checking algorithm to verify triggered conditions. A visual representation of the limit order submission process is shown in Figure 5.

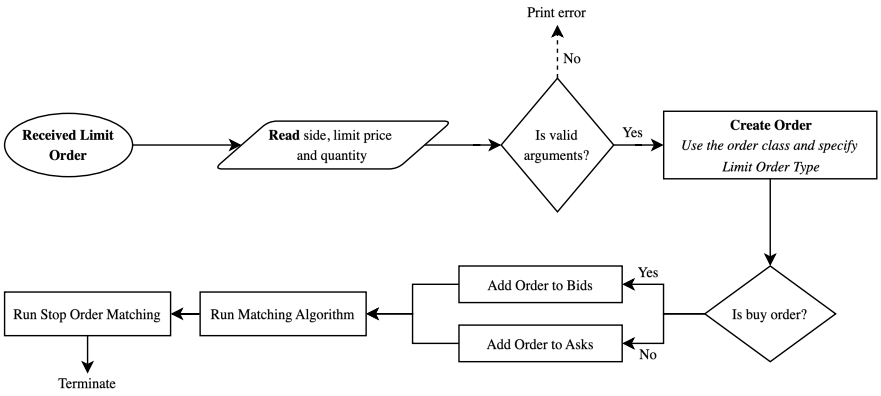

Figure 5: Limit order submission process.

2.4 Stop Orders

Stop orders are conditional orders that activate only when the market reaches a predefined stop price. Once triggered, they convert into either market or limit orders, following the respective submission process and triggering the matching and stop order checking algorithms. Until activation, stop orders are stored separately from active limit orders, ensuring they do not interfere with normal order matching. A visual representation of the stop order submission process is shown in Figure 6.

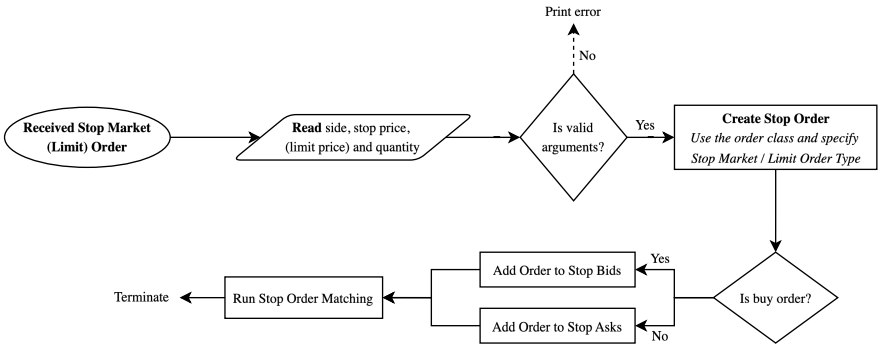

Figure 6: Stop order submission process.

2.5 Matching Algorithm

The order matching algorithm is the core mechanism for trade execution, ensuring orders are matched based on Price-Time Priority—first by the best available price, then by FIFO execution within the same price level.

Triggered upon order submission, the algorithm checks for price crossing ($\exists$ bid : Price(bid) $\geq$ Price(ask)). If a match exists, orders are filled until the condition is violated, at which point the loop terminates. Price-Time Priority is maintained by strictly following the order of the ask and bid maps and executing orders at the front of the list within each price level before proceeding. A visual representation of the matching algorithm is shown in Figure 7.

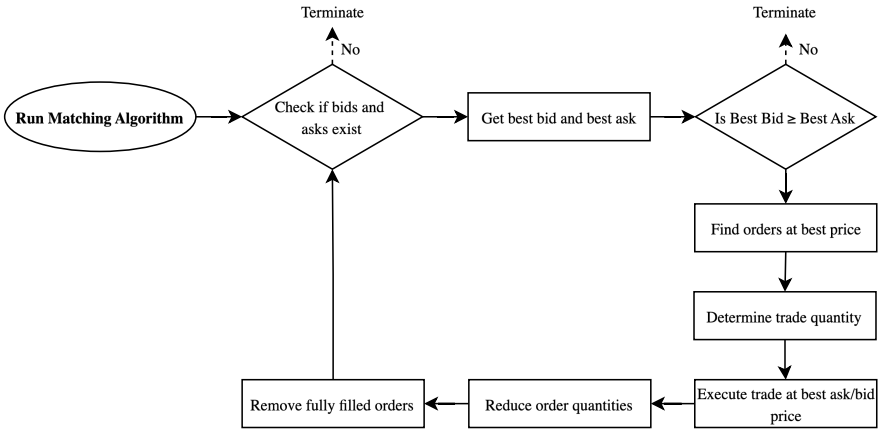

Figure 7: Matching algorithm process.

2.6 Check Stop Orders Algorithm

The check stop orders algorithm determines whether stop orders should be activated, triggering whenever a new trade occurs to evaluate stop conditions in real time against the latest market price. A key design choice is that the algorithm does not iterate over all stop orders. Instead, it checks only the front elements of the stop order maps. If these do not meet the trigger conditions, the rest will not either, ensuring efficient execution.

When a stop order is triggered, it is converted into a market or limit order and submitted to the order book, after which the matching and check stop orders algorithms are re-executed iteratively until no further orders can be processed. A visual representation of the check stop orders algorithm is shown in Figure 8.

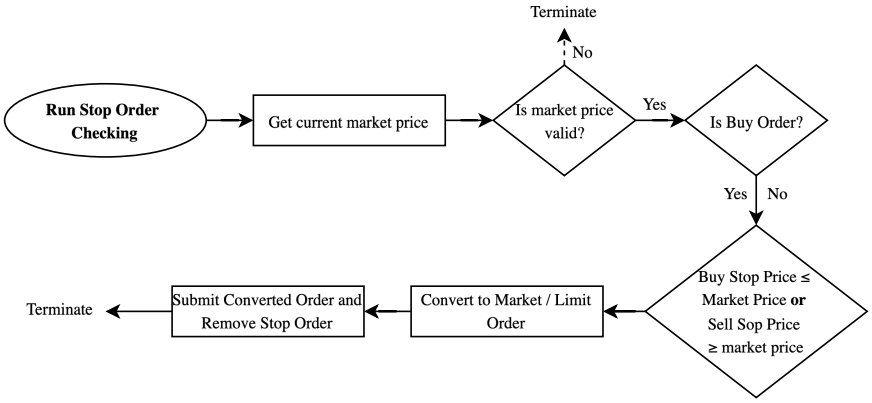

Figure 8: Check stop orders algorithm overview.

3 Discussion

While our C++ Limit Order Book implementation efficiently employs appropriate data structures—such as `std::map` for price levels, `std::list` for FIFO order, and `std::shared_ptr` for memory management—there remain several areas for improvement.

The current in-memory storage mechanism, though sufficient for simulation purposes, would certainly fail under the high-volume conditions typical of real-world trading. In a production environment, persistent storage through a shared database is essential. Such a system would not only facilitate scalability by allowing multiple instances of the order book to access and update a central data repository but also enable access to historical trade data.

Efficiency of the matching algorithm is another area where further optimization is warranted. Although the algorithm correctly enforces Price-Time Priority, its performance in high-frequency trading scenarios could benefit from parallel processing techniques and concurrency management to reduce latency.

Additionally, the current system is designed for a single security, limiting its practical applicability. Extending support to multiple stocks and different types of securities would significantly enhance its utility. This expansion would require the introduction of additional abstraction layers to manage multiple order books concurrently.

Some design choices, particularly regarding data types, also merit reconsideration. While whole-stock trading simplifies the model, it excludes the possibility of fractional share transactions. Furthermore, employing fixed-width integer types (e.g., 32-bit or even 16-bit where appropriate) could ensure consistency and prevent overflow in high-volume settings.

Lastly, our current CLI-based interaction model, while useful for demonstration, is not representative of secure, real-world trading systems. In practice, an API-based architecture would be preferable, allowing safe and flexible access through web or mobile interfaces.

4 Conclusion

Overall, this project serves as a valuable foundation for anyone looking to grasp the mechanics of a limit order book within a C++ framework. The separation into distinct classes for orders, order handling, and order management showcases good software design principles and clarifies each component's role. By meeting the initial aims—namely, to explore order book mechanics and demonstrate a clean object-oriented structure—the project has proved effective as both a learning exercise and a stepping-stone for more advanced trading systems.